DOI:10.1145/3126907

To view the accompanying paper, visit doi.acm.org/10.1145/3126905

Technical Perspective
Shedding New Light on an Old Language Debate

By Jeffrey S. Foster

AS COMPUTER SCIENTISTS, we use programming languages to turn our ideas into reality. It is no surprise, then, that programming language design has been a major concern since at least the 1950s, when John Backus introduced FORTRAN, usually considered the first high-level programming language. The revolutionary innovation of FORTRAN—the thing that made it high-level—was that it included concepts, such as loops and complex expressions, that made the programmer's job easier. To put it another way, FORTRAN showed that a programming language could introduce new *abstractions* that were encoded via a compiler, rather than directly implemented in the hardware.

Not long after the introduction of FORTRAN, other programming languages appeared with somewhat different sets of abstractions: John McCarthy's LISP, which introduced functional programming, and Grace Murray Hopper's COBOL, which aimed to support business, rather than scientific or mathematical, applications. Thus, for at least the last 60 years, programmers have been faced with the question: What programming language should I use?

Today, answering this question has only gotten more difficult. Myriad languages have been developed in the last six decades, with at least a few dozen in common usage today. Moreover, even if in a theoretical sense any general-purpose language can implement any algorithm, in practice the different abstractions provided by different languages seem to have a strong influence on programming tasks. The Internet is filled with heated debates about the merits of functional versus imperative programming, the costs versus the benefits of object orientation, and the trade-offs between dynamic and static typing, among many others.

The following paper aims to bring empiricism to this debate by studying whether programming language choice and code quality are related. To do so, the authors perform an observational study on a corpus of 728 popular GitHub projects, totaling 63 million lines of code. The largest single project considered is Linux, comprising 15+ million lines of code. The authors apply a variety of techniques to extract data from the subject programs to try to answer the question at hand. They rely on GitHub Linguist to identify the primary language of a project. As a proxy for code quality, they count the total number of defect-fixing commits per project, determined by textually searching for one of a fixed set of keywords in the commit messages. And they use supervised machine learning to approximately classify bug reports into categories.

The authors then apply a range of statistical methods and reach four main conclusions. First, they report that projects in Clojure, Haskell, Ruby, and Scala are slightly less likely to have defect-fixing commits, and those in C, C++, Objective-C, and Python are slightly more likely. Second, they report that languages that are functional, disallow implicit type conversion, have static typing, and/or use-managed memory have slightly fewer defects than languages without these characteristics. Third, they report that defect-proneness does not depend on domain, for example, user application versus library versus middleware. Last, they report that some defect types, such as memory errors, are strongly associated with certain languages.

For at least the last 60 years, programmers have been faced with the question: What programming language should I use?

These are interesting results. While they suggest that the language does indeed matter, almost all of the observed effects are small … except that some particular language features, such as a lack of memory safety, do have profound effects.

Like any empirical study, the results here have threats to validity: noise in the data, such as the classification of a commit as defect-fixing, is difficult to account for; defects may have been made and fixed without an intervening commit, for example, defects prevented by a static type checker are likely not included; projects vary significantly in software engineering practices, for example, Linux is an outlier, with an extremely large user base with many developers and testers; tool support for different languages varies significantly; there may be a strong relationship between programmer skill and language choice; language design can obviate classes of errors, for example, buffer overflows can occur in C and C++ but not Java; and in practice the choice of programming language is often constrained both by external factors (for example, the language of existing codebases) and the problem domain (for example, device drivers are likely to be written in C or C++). Finally, while the use of regression analysis in such observational studies can control for confounds and strongly suggest relationships, it cannot definitively establish causation. Even so, this paper raises intriguing questions in an effort to shed light on one of the oldest debates in computer science.

Jeffrey S. Foster (jfoster at cs.umd.edu) is a professor in the computer science department at the University of Maryland, College Park.